[redacted]

1

A TECHNIQUE FOR PROVING SPECIFICATIONS ARE MULTILEVEL SECURE

OK DTIC

10 January 1980

Computer Science Laboratory Report CSL-109

By: Richard J. Feiertag, Computer Scientist
SRI Computer Science Laboratory
Computer Science and Technology Division

SRI International
333 Ravenswood Avenue
Menlo Park, California 94025
(415) 326-6200
Cable: SRI INTL MPK
TWX: 910-373-1246

SRI International

A TECHNIQUE FOR PROVING SPECIFICATIONS ARE MULTILEVEL SECURE

10 January 1980

Computer Science Laboratory Report CSL-109

By: Richard J. Feiertag, Computer Scientist

SRI Computer Science Laboratory
Computer Science and Technology Division

The work described in this report was funded by a variety of sources dating back to 1973. The specification approach began with work on PSOS for the U.S. Department of Defense, and later also received support from NSF and the U.S. Navy (NOSC). The multilevel security (MLS) model has its roots in the work of Bell and LaPadula at MITRE. SRI's involvement with the model began with work on the PSOS secure object manager and work on a proposed multilevei-secure kernel for Multics (DARPA/Honeywell). The present MLS model used here was formulated precisely for the first time in work on TACEXEC for the U S. Army (Ft. Monmouth). The MLS proof tools were developed with several sources of funding, primarily under a subcontract from Ford Aerospace on the KSOS project (sponsored by DARPA), but also with support from the U.S. Army and from Honeywell The tools described here are being used for both KSOS-11 and KSOS-6.

SRI International

Table of Contents

List of Figures

1. Introduction

The following sections describe a technique for verifying that a design for an operating system or subsystem expressed in terms of a formal specification is consistent with a particular model of multilevel security. The technique to be described is mathematically rigorous and, if applied properly, gives assurance that the given design is multilevel secure by this particular model. The technique is supported by a collection of automated tools. These tools assist the user in the performance of a large amount of detailed routine tasks that must be performed to apply the technique. In general, contemporary formal verification techniques such as the one described here involve a great deal of repetitive, detailed, uninteresting steps that are necessary to maintain the rigor of the proof process. The proofs are usually larger and more complex than the system being proved. Under such circumstances, the accuracy of a proof performed manually is highly suspect. Although the accuracy of such proofs conducted using automated tools is also highly suspect, the automation provides the following advantages.

- The results are repeatable. Successful application of the automated tools to a number of system designs increases ones confidence in the tools. This confidence carries over to application of the tools to other systems.

- Use of automated tools significantly increases the speed with which the verification can be accomplished. If the tools are sufficiently fast, then the verification can reasonably become part of the design process.

- Once the development cost of the tools is sufficiently amortized, the cost of applying the technique becomes very small and the verification becomes highly cost effective due to the savings produced by early discovery of errors in the design.

- The application of a rigorous verification technique increases ones confidence in the security of the resulting system, thereby increasing its usefulness to the user. Unfortunately this increased usefulness is realized only if the increased confidence is justified.

This last advantage emphasizes the point that it is important for interested parties to understand what the verification technique described here verifies and what it doesn't verify. The term "proof" is often taken to mean that all problems have been solved. All "proof" techniques make assumptions and have limitations and it is essential that these assumptions and limitations be clearly understood. In the case of the technique described here the following assumptions and limitations apply.

- The multilevel security model described below reflects the desired security for the system in question. Clearly, if the property being verified is not the desired property, then no matter how correct the verification it is not of much use.

- The design of the system, as embodied in the formal specification, correctly describes all aspects of the behavior of the system. There is a great deal of value in having a specification that has been verified as multilevel secure. However, if the implementation of the system does not follow the specification, much of that value is lost.

- The verification technique is correct and mathematically sound.

- The multilevel security model is mathematically sound and consistent.

- The formal specification is mathematically sound and consistent.

- The automated tools correctly embody the verification technique.

All of the above assumptions are more difficult to formally verify than the property that is being verified. In fact, it is the making of these assumptions that allows one to consider the relatively simple property that is being verified. However, this does not diminish the importance of the result. Although it may be hard to formally verify these assumptions, there has been excellent success in informally verifying them. What has been done here is to focus on that part of the problem that has historically been error prone, thus achieving the greatest impact at the least cost. Thus formal verification is seen not as the ultimate solution to the problem of eliminating bugs from systems, but as one very effective technique in reducing certain types of errors.

As with the verification technique itself, successful use of the automated tools requires that the user understand what the tools do and what their limitations are. Even in the case where one gives a specification to the tools and all resulting formulas are proved true, it is important that one understand what is being proved. This depends upon the information that the user has given the tools. Of course, it is not likely that everything will be proven on the first try. The user must be able to understand the error messages produced by the tools and must be able to interpret the output generated by the tools. This will enable the user to modify the inputs to the tools to achieve a more successful result. A fully successful result does not necessarily require that all formulas generated be proved true. Some systems may be considered adequately secure even though they contain some security flaws. In other cases, an unproved formula may not necessarily mean a security flaw and other techniques may be applied to verify the security of the specification. These tools should therefore be viewed as an important aid in verifying the multilevel security of a specification, but they are not, by themselves, necessary nor sufficient to meet this end.

2. Multilevel Security Model

The basic multilevel security requirement is assurance that when information is passed from one entity to another within the system, that the information is being passed to an entity that is at least as secure as the entity that the information is coming from. We can identify in a system two types of identities: subjects and objects. Subjects are the active entities that perform the computations. Examples of subjects include users, processes, and tasks. Objects are repositories, sources, or sinks of data. Examples of objects include files, memory, storage devices, and hard copy. Each subject and object is assigned a security level. The security levels are ordered by a partial ordering relation. The higher the security level in the ordering, the more secure the associated subject or object. The multilevel security requirement states that when information flows from subject to subject, object to object, subject to object, or object to subject, the recipient entity must have a security level that is at least as high as that of the sending entity. In other words, there is no declassification of information.

In normal military security, a security level consists of a classification or clearance and a set of categories. The classifications are totally ordered and have names such as UNCLASSIFIED, CONFIDENTIAL, SECRET, and TOP SECRET. The set of categories is simply a set of predefined compartments. The partial ordering relation for military security states that one security level is lower than or equal to another security level if the classification of the first level is less than or equal to that of the second by the total ordering of classifications, and the set of categories of the first is a subset of that of the second. The formal model described below does not mention classifications or category sets, it simply considers security levels and ordering relations. Therefore, military security is a special case of this model. This in no way modifies the result.

In some cases the multilevel security requirement is overly restrictive for certain applications. In such cases it is meaningful to talk about less restrictive formulations of the model. A few such formulations are discussed briefly towarc the end of this document.

2.1. General Model

A system consists of a collection of operations or functions. Each function may be invoked by a user of the system. (Actually the function is invoked as part of a program running on behalf of a user.) When invoked, a function may take a set of arguments. A function together with a particular set of arguments is termed a function reference. When a function reference is invoked, it can cause the state of the system to change and/or return information to its invoker. The set of all function references of a system is called F and some member of this set is denoted by f.

We also define a set of security levels L. The security levels L are partially ordered by the relation "$\leq$". "$\leq$" is a reflexive partial ordering relation in that it must be reflexive, transitive, and antisymmetric. Multilevel security involving classifications and categories is but one example of a partial ordering of security levels, so we will be dealing here with a more general case. There is a function K whose domain is F and whose range is L. The function K returns the security level of its argument. A process is assigned a security level for its lifetime and may only invoke function references at this level. (Note that a user may have several processes operating on his behalf simultaneously, and may therefore operate at several security levels.)

Finally, we introduce the relation "-->" on function references. We say that

$$f_1 \text{-->} f_2$$

(read as f_1 transmits information to f_2) if there is any possibility that the information returned by an invocation of f_2 could have been in any way affected by a prior invocation of f_1. In other words, there is some transmission of information from f_1 to f_2.

The definition of multilevel security can now be stated simply. For any f_1 and f_2 in F:

$$f_1 \text{-->} f_2 \implies K(f_1) \leq K(f_2) \quad (1)$$

This simply states that if there is any possibility of information transmission between two function references, then the transmitting function reference must have a security level less than or equal to the that of receiving function reference.

In other words, information can flow only upward in security or remain at the same level. (It should be noted that in military multilevel security cases, "the same level" implies <u>identically</u> the same classification and the same categories.) An alternative definition is given by Feiertag et. al. [3].

Unfortunately, the abstract nature of this definition makes it difficult to relate to constructs used in expressing system designs. This gap can be bridged by formulating a slightly more restrictive model in less abstract terms.

2.2. Restricted Model

Each state variable v contains some of the state information of the system. The state variables together completely describe the state of the system. The value of each state variable may be modified by invocation of some function reference. Each state variable is assigned a security level which is determined by extending the function K to apply to state variables as well as function references. Therefore, K(v) is the security level of state variable v. The relation -f> relates two state variables such that

$$v_1 \text{ -f> } v_2$$

means that an invocation of function reference f may cause the value of v_2 to change in a manner dependent upon the previous value of v_1. In other words there is an information flow from v_1 to v_2 caused by the invocation of f. Two predicates must also be defined: the prefix form of -f>

$$\text{-f>}v$$

means that an invocation of the function reference f may cause the value of state variable v to change; the postfix form

$$v \text{ -f>}$$

means that the value returned by function reference f is dependent on the prior value of state variable v. Note that for any f_1, v_1, and v_2:

$$v_1 \xrightarrow{f} v_2 \Longrightarrow \xrightarrow{f} v_2$$

A multilevel secure system may now be redefined. For any function reference f and state variables v, v_1, and v_2

$$v \xrightarrow{f} \Longrightarrow K(v) \leq K(f) \tag{2}$$

$$v_1 \xrightarrow{f} v_2 \Longrightarrow K(v_1) \leq K(v_2) \tag{3}$$

$$\xrightarrow{f} v \Longrightarrow K(f) \leq K(v) \tag{4}$$

These properties assure that information flow is always upward in security level or remains at the same security level. Loosely speaking, the arrow --> always points upward in security level. Equation 2 states that the value returned by an invocation of a function reference at some security level contains information from state variables at only lower or equal security levels. Equation 3 assures that when information is transferred from one state variable to another by some invocation of a function reference, that the recipient variable is at a higher or equal security level than the originating variable. Equation 4 assures that the value of a state variable may be changed by invocation of a function reference whose security level is less than or equal to that of the variable, thereby guaranteeing that security cannot be violated by the act of invoking a function reference. An alternative definition is given by Feiertag et al. [3].

2.3. Formal Definitions of Relations and Predicates

A multilevel system is defined to be the following ordered 9-tuple:

$$\langle S, s_0, L, \text{"}\leq\text{"}, F, K, R, N_r, N_s \rangle$$

where the elements of the system can be intuitively interpreted as follows:

- S - States: the set of states of the system

- s_0 - Initial state: the initial state of the system; $s_0 \in S$

- L - Security levels: the set of security levels of the system

- "$\leq$" - Security relation: a relation on the elements of L that partially orders the elements of L

- F - Visible function references: the set of all the externally visible functions and operations (i.e., functions and operations that can be invoked by programs outside the system); if a function or operation requires arguments, then each function together with each possible set of arguments is a separate element of F (note that in this document externally visible functions and operations will be referred to collectively as visible functions (or functions) even though operations are not functions in the mathematical sense)

- K - Function reference security level: a function from F to L giving the security level associated with each visible function reference; a process may invoke only function references at the security level of the process; K:F->L

- R - Results: the set of possible values of the visible function references

- N_r,N_s - Interpreter: functions from FXS to R and S that define how a given visible function reference invoked when the system is in given state produces a result and a new state; N_r:FXS->R and N_s:FXS->S.

There is also a set of state variables V, each member of which is the set of values can be assumed by that state variable. The set of states S is isomorphic to the cross product of all the state variables $v \in V$.

In order to define multilevel security, the following definitions are useful:

- T - the set of all n-tuples of visible function references or, in other words, all possible sequences of operations $T = F^*$

- M - the function whose value is the state resulting from the given sequence of operations starting at some given state M:SXT->S

- D - the function whose value is the set of state variables whose values differ in the given states D:SXS->V^*

The two relations and two predicates described above can now be formally defined:

$$f_1 \dashrightarrow f_2 \equiv$$

$$(\exists t_1,t_2 \in T)$$

$$N_r(f_2,M(t_2,M(<f_1>,M(t_1,s_0))))$$

$$\sim= N_r(f_2,M(t_2,M(t_1,s_0)))$$

$v_1 \xrightarrow{f} v_2 \ \equiv$

$(\exists s_1, s_2 \in S \mid D(s_1,s_2)=\{v_1\})$

$v_2 \in D(N_s(f,s_1),N_s(f,s_2))$

$v \xrightarrow{f} \ \equiv$

$(\exists s_1, s_2 \in S \mid D(s_1,s_2)=\{v\})$

$N_r(f,s_1) \sim= N_r(f,s_2)$

$\xrightarrow{f} v \ \equiv$

$(\exists s \in S)$

$v \in D(s,N_s(f,s))$

3. Interpreting the Security Model for SPECIAL

In order to prove that a given specification of a system written in the specification language SPECIAL is multilevel secure by the above definitions, it is necessary to define SPECIAL as an interpretation of the restricted model. To do this it is necessary to show how each constituent of the model is represented in a specification written in SPECIAL. Recall that the model is a 9-tuple:

$$\langle S,\ s_0,\ L,\ \text{"}\leq\text{"},\ F,\ K,\ R,\ N_s,\ N_r \rangle$$

In addition to the elements of the model, the notion of state variable (a refinement of the concept of state) will also be described in terms of SPECIAL since it plays a central role in the definition of multilevel security.

The set of function references F is given in SPECIAL by the definitions of the visible V-functions, O-functions, and OV-functions in a specification. Since these function definitions are parameterized, each function reference in the set F corresponds to a function definition of a specification that has been instantiated with particular arguments. In other words, a member of F corresponds to a visible function definition of the specification together with a particular value for each parameter in the function definition.

As described above, the set of states S can be defined in terms of the values of a set of state variables V. In a SPECIAL specification it is easy to represent the set of state variables. Each state variable (i.e., each member of V) corresponds to a primitive V-function reference in a specification. A primitive V-function reference is simply a defined primitive V-function together with a value for each of its formal parameters. The collection of current values of all the primitive V-function references therefore defines the current state of the system.

The set of return values R is simply the set of binary tuples whose first element is a natural number and whose second element can be any value defined in the specification. The value of the first element of the binary tuple indicates the first exception whose value is true (0 indicates all exceptions are false) in the function reference returning the value. The latter element of the binary tuple includes not only those primitive values defined by the SPECIAL language, but also values defined by the specifications in the TYPES paragraph. Note that if the first element of the tuple is nonzero, the second element must have undefined value.

The interpretation functions N_r and N_s are represented by the syntax and semantics of SPECIAL. The syntax and semantics of SPECIAL define the value returned by and the new values of the primitive V-function references that result from the invocation of a visible function reference. These values are expressed as a function of the parameters to the visible function reference and the prior values of the primitive V-function references.

The initial state s_0 is determined by the constraints placed on the initial values of the primitive V-functions as given in the INITIALLY clause of a primitive V-function definition. Actually, these constraints do not necessarily constrain the system to a single initial state, but to some set of initial states. This requires that multilevel security be proved for the entire set of initial states. In actuality the proof is carried out for every possible state as the initial state with no increase in difficulty. This simply indicates that the initial state has no relevance to multilevel security as defined above.

The remaining elements of the model, the set of security levels L, the partial ordering function "$\leq$", and the security level function K, are not given as part of the specification but are introduced into the proof process separately. A discussion of how these elements are introduced is given subsequently.

4. The Proof Process

With the interpretation of the model in terms of SPECIAL given above, it is a fairly straightforward process to prove that a given specification written in SPECIAL and following the above conventions is consistent with the model. One simply must identify all instances of primitive V-function references in the specifications that satisfy the relations or predicates given above in Section 2.2 and then show that Equations 2, 3, and 4 are satisfied in each case. In general, it is not decidable whether any of the relations of Section 2.2 is satisfied so, in order to be sure the approach is sound, we will always assume a relation holds if we cannot determine otherwise. In this way we may end up trying to prove that Equations 2, 3, and 4 hold when it is unnecessary to do so, but this is better than failing to prove that these equations hold when we should have done so.

5. Automation of the Proof

The steps described above for accomplishing a proof are sufficiently straightforward that they lend themselves well to automation. Automating the proof process serves not only the purpose of making the proof easier, but in some ways making the proof more reliable and convincing. The proof process requires tedious attention to detail. One must examine each primitive V-function reference of the specifications and prove one or more properties about that reference. Although this is not hard in most cases, it is the type of repetitive, painstakingly detailed work that can easily lead to error. Unfortunately, a simple small error could partially invalidate the proof. To the extent that automation is possible, it can alleviate these problems. Of course, the automation is not necessarily foolproof. It does have the advantage of being accurately repetitive. After demonstrating correct operation on a number of specifications, one can have a great deal of confidence in its continued correct operation if it is not modified. This cannot be said of manual proof.

The remainder of this section discusses the automated system developed for the purpose of pro`ing that specifications written in SPECIAL are consistent with the model given above. For purposes of automation, the proof process is divided into the following parts:

1. Parsing the SPECIAL specification.
2. Checking the specification for consistency in the types of the identifiers.
3. Obtaining the interpretation for those aspects of the model not already given an interpretation, namely, the set L, the relation $\leq$, and the functions K and I.
4. Resolving external references of the module under consideration.
5. Building a symbol table of all identifiers.
6. Renaming of some identifiers to avoid potential future name conflicts.
7. Performing some syntactic transformations which serve to simplify the language of the specification.
8. Identifying all primitive V-function references.
9. Identifying occurrences of the relations given in Section 2.2.
10. Composing formulas using Equations 2, 3, and 4.
11. Simplification of the composed formulas.
12. Proof of the simplified formulas.

These parts of the proof process are not necessarily performed in the order given above and are not necessarily performed as separate steps in the automating programs. Some are performed as separate passes over the entire specifications and some are performed as separate passes over single expressions and parts of expressions.

In this proof process, three of the parts are particularly difficult, requiring heuristic, rather than simple algorithmic techniques. These three are:

- Identifying occurrences of the relations given in Section 2.2.
- Simplication of the composed formulas.
- Proof of the simplified formulas.

The immediately following discussion describes how the automation accomplishes the parts of the proof process other than these three. The difficult three are discussed separately after that.

The automation tools are divided into three main parts as shown in Figure 5-1.

Part 1 is the SPECIAL specification checker of the HDM tools. This part performs the parsing and syntax and type checking of the SPECIAL specifications. The output of this part of the tools is a parse tree of a legal SPECIAL specification. This part of the tools is documented in the HDM Handbook [4] and is not discussed further in this document.

Part 2 is the formula generator which takes the parse tree produced in Part 1 and generates a list of partially simplified formulas whose truth implies the security of the restricted model descrited above. The operation of this part of the automation tools is discussed in detail below.

Part 3 is the theorem prover. The theorem prover used here is a slightly augmented version of the Boyer-Moore theorem prover described in A Computation Logic [1] and whose use is described in "A Theorem-Prover for Recursive Functions: A User's Manual" [2]. Subsequent sections of this document will describe the augmentation of the Boyer-Moore theorem prover used in these tools. It will be assumed that the reader is familiar with the basic theorem prover.

6. The Formula Generator

This section presents a detailed description of the inner workings of the formula generator, Part 2 of the automated multilevel security proof tools for specifications written in SPECIAL. The purpose is to give the reader a start in understanding how the code of the tools functions. The formula generator is written in INTERLISP [5] and the reader of this section should be familiar with this language.

6.1. Important Data Structures

Before discussing the algorithms used in the formula generator, some of the important data structures are described -- namely the following:

- Specification parse tree

- Symbol table

- Symbol list

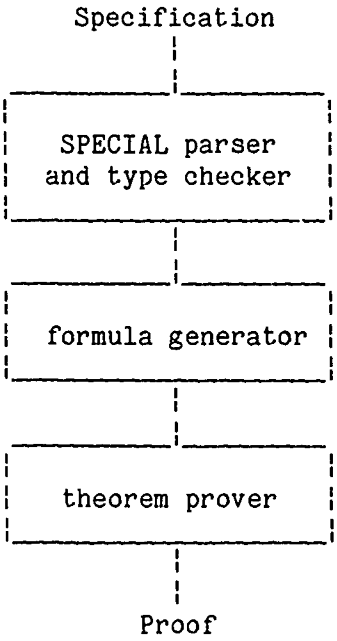

Figure 5-1: Major parts of verification tool

- Transformed parse tree
- SPECIAL operations list
- SPECIAL symbol translation list
- Declared functions list
- Global assertions list
- Local preconditions list
- SOME variables list
- Delayed effects list
- Formulas list
- Theorem prover command list
- Link file
- Special symbols

All of these data structures, with the exception of the link file, exist only while processing a module. The link file persists indefinitely and is used in processing other modules. The purpose (and in some cases the structure) of each of these data structures will be described in turn.

The <u>specification parse tree</u> is the primary input to the formula generator. It is the representation of the specification in tree form. In LISP this turns out to look like a prefix form of the SPECIAL specification from which it is derived by the parser of the HDM tools. The formula generator assumes that the specification is syntactically correct and that all type checking has been performed. The formula generator and theorem prover pay no attention to types.

The <u>symbol table</u> (bound to the variable TABLE) is the central repository of needed information about each identifier encountered in the specification. Each entry in the symbol table contains three standard pieces of information about an identifier plus some additional information whose nature depends upon the nature of the identifier being described. The three standard pieces of information are:

- the name of the identifier in the original specification,
- the name of the identifier to be used in the generated formulas, and

- the nature of the entity to which the identifier corresponds.

The second piece of information is needed because within a given specification, two different identifiers might have the same name. This can happen if the two identifiers appear in different scopes. Because the formulas to be generated do not have scopes as such and because two identifiers with the same name could both appear in the same formula, it is necessary to disambiguate the names. However, it is desirable for someone examining the formulas to be able to relate the name of an identifier in a formula to the name of the identifier in the original specification. Therefore, the renaming is done as follows:

1. The first occurrence of a given name in the specification will be renamed to itself. This means that if a name is associated with only one identifier then the original name will be associated with that identifier.

2. When an identifier is encountered with a name that is already associated with another identifier, and if there is a possibility of generating a formula that references both identifiers, then this most recently encountered identifier will be renamed and the new name will consist of the original name concatenated with a period and some string that will serve to disambiguate. Therefore, the new name will always have the original name as an easily identifiable initial substring. The added string is simply a number assigned sequentially so that the first duplicate name will have the string ".1" concatenated and the next duplicate of the same name will have ".2" added and so on. It is sometimes possible for an identifier name to have several of these disambiguating suffixes. This arises if a definition or function is invoked that has local identifiers that have previously been disambiguated. For example, a definition might have a local identifier arising from the use of the FORALL construct such as: FORALL INTEGER i ¦ P(i): Q(i). If when the definition was first defined the use of the name "i" was ambiguous, it might have been renamed to "i.1". When this definition is invoked in some function description, and if the name "i.1" is already in use in this context, the identifier might get renamed to "i.1.1". Therefore, the number of added suffixes is a clue to the depth of nesting of invocation of the identifier in question.

The third field of the symbol table gives the nature of the entity to which the identifier refers. Possible values for this field include: CONSTANT (an element of an enumerated type), PARAMETER (a parameter of the specification), VFUN (primitive V-functions), OVFUN (derived V-functions, O-functions, and OV-functions), INSTANTIATION (an identifier instantiated in a FORALL, EXISTS, SOME, LET, etc.), ARGUMENT (a formal parameter to a function or definition), DEFINITION, or RETURNARG (a return value in a function). For the cases

CONSTANT, ARGUMENT, and RETURNARG there is no additional information in the symbol table. For the other cases the additional information is as follows:

- PARAMETER - The fourth field is ignored and the fifth field contains a list of the formal argument identifiers to the parameter function. If the parameter is not a function, then the fifth field is empty.

- INSTANTIATION - The fourth field is usually empty, but in the case where the identifier was instantiated from a SOME construct, the fourth field contains the qualifier in the SOME construct. This is necessary because it is important to ascertain if two instantiated identifiers represent the same value. In the case of the SOME construct this can be done by comparing the qualification expressions.

- VFUN - The fourth field is the numeric position of the argument to the V-function that is the security level of that V-function. The fifth field is the formal parameters to the V-function. The sixth field if the identifier representing the return value. The seventh field is the name of the module in which this V-function was defined, and the eighth field is the body of the V-function.

- OVFUN - The fifth through eighth fields of entries for VFUN above become the fourth through seventh fields here. For VFUNs and OVFUNs the body field consists of a list of paragraphs. Each paragraph is a list whose first member identifies the type of paragraph (ASSERTIONS, EXCEPTIONS, DERIVATION, or EFFECTS), whose second member is a list of the local variables of the paragraph, whose third member is a list of the local variables derived from SOME constructs together with their qualifications, and whose fourth member is a list of the expressions in the paragraph.

- DEFINITION - The fourth field is a list of formal parameter identifiers of the definition. The fifth field expression defining the value of the definition. The sixth field is the list of local variables in the definition.

The information in the symbol table is also stored on the property lists of the atoms whose names correspond to the second field in the symbol table. The name of the property under which this information is stored is TGINFO. Placing the information from the symbol table on the property list of the renamed name makes it fairly easy to retrieve the information from the renamed name. Symbol table accesses using the renamed name are far more common than accesses using the original name.

The symbol list (bound to the variable SYMBOLCOUNT) is simply the list of all identifiers that have already been encountered and the number of disambiguated names generated from this name. This list is used when generating suffixes for disambiguation.

As each expression in the parse tree is processed, it is transformed in two ways. The unambiguous names replace the ambiguous names. Certain constructs are replaced with other equivalent constructs. The result of these transformations is the transformed parse tree. This transformed parse tree never exists in its entirety, rather small portions of the entire tree exists as attention is placed upon a specific portion of the tree. However, it is an important conceptual data structure because most of the code acts on the transformed parse tree rather than on the original parse tree. Some of the transformations are these:

- Implications are transformed to conditionals.

- FORALL quantification is transformed into simple expressions in which the previously quantified identifiers become free identifiers.

- Identifiers instantiated in SOME expressions become simple instantiated variables. Vector constructors become universally quantified expressions.

- Definitions, uses of derived V-functions, and EXCEPTIONS_OF and EFFECTS_OF expressions are expanded.

The purpose of these transformations is to reduce the number of different kinds of expressions that subsequent code has to deal with.

The formula generator is basically a syntax driven tool. As an expression is processed, routines are invoked that correspond to the operation in the expression that is currently being processed. The SPECIAL operations list (bound to the variable OPLST) associates the name of a routine in the tools that is called whenever a particular operation in the specification is encountered.

The formula generator converts all special mathematical symbols such as =, <, >, +, etc. to function names. The correspondence between a symbol and its name is given in the SPECIAL symbol translation list (bound to the variable OPNAMES).

All functions which appear in the generated formulas must be declared to the theorem prover. These names are collected on the declared functions list (bound to the variable DECLLIST) and passed on the the theorem prover as part of the theorem prover command list.

The global assertions made in the specification are passed on to the theorem prover where they are defined as axioms. The global assertions are collected on the *global assertions list* (bound to the variable COLLECTABLES) and then passed to the theorem prover as part of the theorem prover command list.

As a function specification is processed, expressions in the EXCEPTIONS, ASSERTIONS, and DELAY paragraphs become preconditions for other parts of the function specification. These preconditions are collected on the *local preconditions list* (bound to the variable PRECONDS) for use when they are needed.

The names of identifiers instantiated in a SOME construct are collected into the *SOME identifiers list* (bound to the variable SOMEVARS). Whenever a new identifier is about to be instantiated in a SOME construct, its qualification is compared with identifiers on the SOME identifiers list and, if the qualifiers are the same, the existing identifier is used and no new identifier is instantiated.

When processing the EFFECTS_OF an OV-function, some of the effects of that OV-function cannot be instantiated at the point of the EFFECTS_OF expression. The instantiation of these effects must be delayed until a place appropriate for their instantiation is encountered. These effects are stored on the *delayed effects list* (bound to the variable DELAYED.EFFECTS) and are instantiated and removed from the list at some subsequent time.

The objective of the formula generator is to generate a list of formulas whose truth implies the multilevel security of the specification. This list of formulas is generated by means of a *formula list*. Each entry on the formula list has three fields. The first field is the formula being generated. At different stages during processing a specification this formula may be only partially complete. The second field is a list of important identifiers appearing in the consequent of the formula. This information is used in the simplification. The third field indicates whether the consequent of the formula is complete. At the time a formula is first generated, it is possible that not all the names of the identifiers in the consequent are known. In this case a temporary name is used to indicate that some identifier is unknown. At a later stage, when the identifier is known, it will be filled in in place of the temporary using the information in the third field.

The information produced by the formula generator must be passed on to the theorem prover (unless a formula can be trivially proven true by the formula generator). This information is passed on to the theorem prover as a list of commands to the augmented theorem prover. The commands have the form of command name followed by some number of arguments. The following is a list of the commands:

- PARTIAL.ORDERING - This command identifies the partial ordering function, "≤", and the security levels at the top and bottom of the partial ordering if such levels exist. The first argument is a list of arguments to the partial ordering function in the order of lower argument first. The second argument is the partial ordering function together with its arguments. The first two arguments essentially identify the partial ordering function and determine if it is a less than or equal to or a greater than or equal to function. The third argument to the partial ordering command is the bottom security level in the partial ordering, and the fourth argument is the top security level.

- SETVARS - This command takes an arbitrary number of arguments. Each argument is a pair, the first element of this pair being the name of a theorem prover variable and the second element of the pair being a value. The theorem prover variable is set to the given value. This command allows the formula generator to set the value of theorem prover variables.

- DECLARATIONS - This command takes an arbitrary number of arguments. Each argument is a function invocation. These function invocations are used to declare the functions to the theorem prover. These functions are derived from the function declaration list described above.

- ASSERT - This command declares as an axiom each of its arguments.

- FORMULA - This command is different in format from those above. The command name does not appear in the command. The first element of the command is a list of information about the origin of the formula. This information is the name of the module, the name of the function, the type of the expression, and the expression from which the formula was generated. The remaining elements of the command are generated formulas. These formulas are successively more complex versions of the same formula. In other words one of the formulas generated by the theorem prover is simplified in successive stages. The various simplified formulas are then placed in the formula command starting from the most simplified and proceeding to the least simplified.

The *theorem prover command list* is placed in a file and this file is read by the theorem prover. The file is called the *formulas file*.

The formula generator processes each module of a specification individually. However, some modules of a specification may reference other modules of the specification. Such cross module references may require that information about other modules be known to the formula generator when it is processing some module. This information is kept in a link file. The link file for a given module is generated by the formula generator when it generates the formulas for that module. This link file can then be read by the formula generator as needed when processing other modules. The primary data structure of the link file is a subset of the symbol table for the module from which the link file was produced. Preceding the symbol table in the link file is some information about the specification source file from which it was produced. This added information helps the formula generator verify that the link file corresponds to the current source file.

There are in addition to the above several items of data that assist the operation of the formula generator. These are given below listed by their INTERLISP variable names:

- LTEQ.FUNCTION - An invocation of the partial ordering function "$\leq$" for security levels.

- BOTTOM.ELEMENT - The name of the least element in the partial ordering of security levels if such a least element exists.

- TOP.ELEMENT - The name of the greatest element in the partial ordering of security levels is such a greatest element exists.

- DEFAULT.LEVEL.NAME - The name supplied as the argument to a specification function that corresponds to the security level of the function if no such name is supplied by the user.

- DEFAULT.TLINK.ANSWER - If the user does not respond to the question about whether or not a link file should be created, this value is used as the reply.

- DEFAULT.WAIT.TIME - The amount of time the formula generator will wait for an answer from the user before using a default response. If the value of this variable is NIL then the formula generator will use the default immediately.

- DEFAULT.LEVELS - A list of the specification function names and the name of the argument to each function that corresponds to the security level last supplied by the user. These argument names are used as the default whenever a function name on the list is again encountered. This makes it possible to utilize the tools repeatedly for the same modules without having to enter the security levels each time.

- CRITICAL.NAMES - A list of names whose presence in the antecedent to a theorem causes the simplifier to include the antecedent in the theorem.

The value of each of these variables can be recorded permanently in a profile file for use in a later session.

6.2. Important Procedures

The formula generator is driven essentially from the transformed parse tree. Starting with the root node of the parse tree, the formula generator examines the name of this node, takes some appropriate action, and then recursively invokes itself for each immediate descendant node. In this way the formula generator eventually encounters all the nodes of the parse tree. (In actuality, some nodes are never seen because there are certain nodes below which descent is not permitted. However, such nodes are not relevant to formula generation.) The LISP function that performs this recursive descent is named TG. The primary function of TG is to examine the name of the node it is given and to call another LISP function to process that node. It obtains the name of the function to call be searching the SPECIAL operations list described above. TG recognizes certain cases where processing is unnecessary because the tree below the node is not relevant, or because the current node is a leaf in the tree and processing is not necessary.

The remaining functions of the formula generator can be divided into three classes: functions supporting dependency heuristics, functions supporting simplification heuristics, and other functions. Functions of the last class are described first. Those of the first two classes are then described next.

- ADD.FUNCTION - This function adds an entry to the SPECIAL operations list. The first argument is the name of the SPECIAL operation to be added to the list and the second argument is the name of the INTERLISP function to be called when that operation is encountered. If no second argument is supplied, the function TGNARY will be called.

- ADDPRECOND - The only argument is an expression to be added to the local preconditions list or the global assertions list, depending on context.

- ADDSYMBOL - The first argument is the name of a newly encountered identifier which is added to the symbol table. The second argument is a list of fields which become fields of the symbol table entry beginning with the third field. ADDSYMBOL returns the disambiguated name for the identifier.

- ASSERTIONS - This function is called by TG to process all assertions paragraphs whether they be local or global. Global assertions are added to the global assertions list. Local assertions are added to the local preconditions list. All assertions are transformed before being added to the appropriate list.

- CLOSETHMS - This function is called to generate formulas for some top level expression. A formula list generated from the given expression is returned.

- COMPLETETHMS - This function is called to resolve any previously unresolved consequents in a formula list. Its first argument is the list of formulas to be resolved and the second argument is a list containing one security level which is used to resolve the incomplete consequents in the formulas. COMPLETETHMS searches the formula list for unresolved formulas (which can be identified by a non-NIL third field) and substitutes the security level given in the second argument for the temporary used to hold the place for the level. The resolved formula list is returned.

- CONSTLIST - This function is a predicate of two arguments. The first argument is an expression. The second argument is a flag. If the flag is NIL, CONSTLIST is true if the expression contains no V-functions. If the flag is non-NIL then CONSTLIST is true if the expression contains no modified V-functions.

- DEFINE.EXTERNALS - This function is used to read and process a link file from another module referenced by the module currently being processed. It reads the symbol table from the link file and adds the symbols to the current symbol table if they are needed by the current module.

- DEFINITIONS - This function is called by TG to process all definitions paragraphs in the specification. The definitions are transformed and are added to the symbol table. Before processing, the definitions are sorted so that all definitions are defined before being used in other definitions. Any recursive definitions or mutually recursive definitions are ignored and a warning message is printed.

- DEFINITIONS.SORTED - This functions is a utility routine of the DEFINITIONS function and actually does the adding of definitions to the symbol table.

- DERIVATION - This function is called by TG to process the derivation paragraph of a V-function. The derivation is transformed and formulas are generated. The resulting formula list is returned.

- EFFECTS - This function is called by TG to process the effects paragraph of an O-function or OV-function. Each effect is transformed, and formulas are generated. The concatenation of the formula lists from all the effects are returned.

- ENCLOSE.AND - This function is a utility routine which takes as an argument a list of expressions to be conjoined. The returned value is an expression which represents the conjunction.

- EXC.THMS - This function is a utility function of the EXCEPTIONS function It takes a single transformed exception and returns the formula list generated from the exception. The negation of each exception is added to the local preconditions list.

- EXCEPTIONS - This function is called by TG to process the exceptions paragraph of a function specification. Each exception is transformed and formulas are generated. The negation of each exception is added to the local preconditions list. The returned value is the list of all formulas generated for the paragraph.

- EXTERNALREFS - This function is called by TG to process the external references paragraph of a specification. Any data required from other modules is obtained by reading the link files of the other modules. The data read is added to the current symbol table. The names of all files read is printed.

- EXTRACTVAR - This function is a utility function that is given an identifier specification and returns the identifier name. An identifier specification may be simply the identifier name or the identifier type followed by the identifier name.

- FIND.SYMBOL - This predicate is true if the first argument is contained in the second.

- FUNCTIONS - This function is called by TG to process the functions paragraph of a specification. The functions are ordered so that all function specifications are defined before they are referenced. Any recursive or mutually recursive function specifications are ignored and a warning message is printed. Each function is processed and its definition is added to the symbol table. If the function specification causes any formulas to be generated, the necessary commands are added to the theorem prover command list and written into the formulas file.

- GENERATE.DCLS - This function generates the DECLARATIONS command for the theorem prover command list.

- GENERATE.FORMULAS - This is a high-level function that calls the parser, the type checker, and the formula generator for a given specification contained in the file whose name is given as its first argument. The third argument indicates one of several possible debugging modes. The fourth argument, if non-NIL, indicates that the formulas file should be created in TOPS-20 temporary mode.

- GET.DEFAULT.LEVEL - This utility function returns the name of the default security level associated with the given function name.

- GET.LTEQ.FUNCTION - This utility function prompts the user for the partial ordering function, the greatest element in the partial ordering, and the least element in the partial ordering. If the user does not respond then the default values are used. These data are stored in the variables LTEQ.FUNCTION, TOP.ELEMENT, and BOTTOM.ELEMENT.

- GETLEVEL - This function returns the security level associated with a given specification function.

- GETPRECONDS - This function returns the expressions on the local preconditions list.

- GETSYMBOL - This function returns the symbol table entry for the given identifier name.

- GT - This is a high-level function which sets up the debugging attributes of the formula generator, calls TG with the entire module specification, and writes out the link file if requested to do so.

- INSTANTECEDENTS - This utility function takes a list of antecedents as its first argument and instantiates these antecedents in each formula in the formula list given as its second argument.

- INSTDEF - This utility routine expands a definition. The first argument is the list of formal parameters to the definition, the second argument is the list of arguments to the definition, the third argument is the body of the definition, the fourth argument is the list local identifiers of the definition, and the fifth argument is the list of identifiers together with qualifications of all local identifiers of the definition that were instantiated in a SOME construct. The returned value is the body of the definition in which the arguments have been substituted for the formal parameters and where all the local identifiers have been suitably renamed to avoid ambiguity.

- INSTMODVFUN - This function generates the necessary formula whenever a modified V-function or return value identifier is encountered. The presence of a modified V-function indicates the truth of a relation of the form $\xrightarrow{f} v$, and a formula of the form in Equation 4 is generated. The existence of the modified V-function or return value identifier in the expression is noted for use when an unmodified V-function is encountered.

- INSTVFUN - This function generates the necessary formulas whenever an unmodified V-function is encountered. The presence of an unmodified V-function indicates the truth of a relation of the form $v_1 \xrightarrow{f} v_2$ or $v \xrightarrow{f}$, and a formula of the form in Equations 2 or 3 is generated. If a modified V-function or return value identifier has not already been encountered in the expression being processed, a temporary identifier will be created to hold the place for the security level of the modified V-function or return value identifier. The function COMPLETETHMS will later replace the temporary with the needed security level.

- LEVELARG - This utility function is invoked to request from the user the name of the argument corresponding to the security level of the given specification function. The first argument to LEVELARG is a list of argument names to the specification function. The second argument is the name of the specification function. The third argument is a list of other answers which LEVELARG can accept from the user. The return value is the answer provided by the user or, if the user did not respond, the default response.

- MAINEXP - This utility function takes an expression, a security level, and data about the expression and returns a theorem prover command list of formulas generated from the expression. The security level is the level of the function being processed .or use in generating formulas for exceptions. In addition, MAINEXP does simplification of the formulas, but this will be discussed in a subsequent section.

- MAKE.PROFILE - This high-level function produces a profile file whose name is the given argument. The profile file contains the values of certain formula generator variables. If no argument is provided, the name MLS.PROFILE is used.

- MAKTHM - This utility function takes as its first argument a list of antecedents and as its second argument a consequent, and returns an expression for the desired implication.

- MAKEXP - This utility function takes an expression and returns the transformed version of the expression.

- MODULE - This function is called by TG with the entire module to be processed. It sets up the environment in which the rest of the processing is to be done and declares an error handler to catch any errors that could not be handled by other functions.

- NILFUN - This function returns the empty list. It is called by TG to handle certain subexpressions that do not generate any formulas.

- NORMQUANT - This function takes the qualification part of many expressions (e.g. FORALL, LET, EXISTS) and returns two lists. One list is simply the identifiers instantiated in the qualification and the other is a list of qualification expressions.

- OFUN - This function is called by TG to handle O-function definitions. It queries the user as to the security level of the O-function, returns a theorem prover command list of formulas generated from the O-function definition, and puts the O-function onto the symbol table. It also prints out statistics about the formulas generated.

- OVFUN - This function performs the same duties as OFUN, but for OV-functions instead of O-functions.

- PAIR.PERMUTATIONS - This utility function takes as arguments a list and a relation. It generates all pairs of elements in the list that obey the relation. This function is used for sorting of definitions and specification functions.

- PARAGRAPH.ORDERP - This predicate indicates whether two paragraphs of a specification function definition are in the proper order for processing.

- PARAMETERS - This function is called by TG to process the parameters paragraph of a specification. The parameters are added to the symbol table and to the function declaration list.

- PARTIAL.SORT - This utility function takes as arguments a list to be sorted and a list of pairs generated by applying PAIR.PERMUTATIONS to the list. The result is a list sorted according to the partial ordering relation given to PAIR.PERMUTATIONS.

- PRUNE - This function takes a list of antecedents and a list of names and returns a list of the antecedents which form the closure on the names. This guarantees that none of the names in the second argument to PRUNE and none of the names in the list of antecedents returned by PRUNE appear in those antecedents in the first argument to PRUNE but not returned by PRUNE.

- PRINTEXP - This utility function prints its argument.

- READ.PROFILE - This high-level function reads in and sets the values of the variables in the profile file whose name is given as its argument. If no name is given, the name MLS.PROFILE is used. A READ.PROFILE is automatically executed each time the tools are initialized.

- READARG - This utility function is called to query the user for a file name if none is given where needed in a user command.

- RELVARS - This utility function takes an expression and returns a list of important identifiers in that expression.

- RETURNARG - This utility function takes a formal argument list and returns the identifier representing the return value of that argument list.

- SET.DEFAULT.LEVEL - This high-level function takes a function name and an argument name, and makes the argument name the default security level for the function.

- SOMECOMPARE - This utility function takes an identifier to be instantiated in a SOME construct and returns an identifier representing an equivalent value if one exists.

- SOMEVARS - This utility function takes a list of identifiers and returns a list of those identifiers instantiated in SOME constructs together with their qualifiers.

- STRIPARGS - The utility function takes a formal parameter list and returns a list of the formal parameter names.

- STRIPTYPE - This utility function takes an identifier specification and returns a list containing the identifier name.

- TGDELAY - This function is called by TG to process the DELAY paragraph in a specification function definition. Formulas are generated for the UNTIL clause. The UNTIL clause becomes a precondition for the effects of the specification function, and the negation of the UNTIL clause becomes a precondition for the effects in the WITH clause.

- TGEFFECTS_OF - This function is called by TG to process an EFFECTS_OF expression. If the function being expanded is an O-function, then the effects of the target O-function are suitably renamed and formulas are generated for them. If the function being expanded is an OV-function, then -- after renaming -- formulas are generated for the effect in the target OV-functions containing the return value identifier; generation of formulas for the other effects is delayed until an appropriate situation is encountered.

- TGERROR - This utility function prints an error message and signals an error.

- TGNEWSTATE - This function is called by TG to process modified V-functions. The actions taken are described under INSTMODVFUN.

- TGRENAME - This utility function takes an expression and transforms it. All identifier names are disambiguated, and certain constructs are converted to equivalent constructs.

- TGSELECT - This utility function takes a list of expressions, a list of truth values, and a truth value. It returns a list of all those expressions in the first list that correspond to values in the second list whose truth value matches the third argument. In other words TGSELECT is a filter.

- TGTYPES - This function is called by TG to process the types paragraph of a specification. The only information in the types paragraph of interest to the formula generator is those identifiers which are declared elements of enumerated types. Those identifiers are entered in the symbol table as constants.

- VARARGS - Certain SPECIAL operators take an indefinite number of arguments. However, the theorem prover does not accept such operators as functions. This utility function takes expressions and (if they contain such an operator) renames the operator to a distinct function name depending on the number of arguments. The result is a transformed expression containing functions that take only fixed numbers of arguments.

- VERIFIY - This high level function calls the parser, type checker, formula generator, and theorem prover with a given file containing a specification to be processed. This is the top level function invoked by the user.

- VFUN - This function is called by TG to process V-function definitions. It performs the same duties as OFUN except for V-functions instead of O-functions.

6.3. Dependency Determination Procedures

For each operator that can appear in an expression, there is an INTERLISP function (called by TG) to determine if the value of that expression is dependent upon the values of the arguments of the expression. For most cases the answer to this question is a simple "yes", and any formulas generated from the arguments are simply merged into a single formula list with this list being returned (to TG) as the result of processing the expression. However, in some cases the answer is not a simple "yes" and more complex processing is done. For example, in a conditional expression, the value of the expression is dependent upon the arguments to the expression only if certain conditions are met. In this case these conditions are added as antecedents to the formulas generated from the arguments, and the resulting list is returned. The following describes what is done for each SPECIAL operation. The name given in parentheses following the operation is the INTERLISP function called by TG to do the processing and the dependency decision making. Some of the SPECIAL operations are not included explicitly because certain operations are transformed away during the transformation process. A few SPECIAL operations are not implemented (and are listed in a subsequent section).

- AND (TGAND) - Formulas are generated for each of the arguments. Formulas are generated for any delayed effects and concatenated to the formula list. If any of the arguments are constant expressions (i.e., do not contain any primitive V-functions) then they become antecedents to the formulas generated from the other arguments. This latter step is useful because if any of the arguments of a conjunction is false, then the conjunction is false, the values of the other elements of the conjunction are irrelevant, and there is no dependency upon them.

- . (TGDOT) - The value of a "." operator is dependent only upon the value of its left argument. The right argument is a field name and a constant.

- IF, =>, EXISTS (TGIF) - The value of an IF operator is dependent upon either the value of the THEN clause or the ELSE clause depending upon whether or not the boolean expression is true or false. Therefore, the boolean expression is added as an antecedent to all formulas generated from the THEN clause, and the negation of the boolean expression is added as an antecedent to all formulas generated from the ELSE clause. The formulas generated from the two clauses are concatenated with the formulas generated from the boolean expression itself. Implication is treated as an IF expression with no ELSE clause. Existential quantification is treated as implication. This latter case of existential quantification is a case where the dependency rule is overcautious, since the value of the existentially quantified expression need be dependent upon the body of the existential quantifier for only one value of the quantified identifier. However, the tools have no easy way of determining which value that is, so the worst case assumption is taken in assuming dependency for all values of the quantified identifier.

- LET (TGLET) - The qualifications in the LET expression become antecedents to formulas generated from the body of the LET expression. The formulas generated from the body are concatenated with the formulas generated from the qualifications.

- OR (TGOR) - The formulas generated from the arguments are concatenated with the formulas generated from any delayed effects.

- {}(qualified form) (TGSET) - The formulas generated from the qualifier are returned.

- VECTOR(FROM-TO form) (TGVECTOR) - The expressions in the FROM and TO clauses become antecedents to the formulas generated from the body of the VECTOR expression. The formulas generated from the body are concatenated with the formulas generated from the FROM and TO clauses.

- CARDINALITY, DIFF, =, FRACTPART, INSET, INTER, .., INTPART, LOG, MAX, MIN, MOD, ~, [], {}(itemized form), -, /, ^, >, <, >=, <=, ~=, +, *, UNION, STRUCT, SUBSET, SUM, VECTOR(itemized form) (TGNARY) - The formulas generated from each of the arguments are concatenated and returned. If more than one modified V-function is encountered in the expression, an error is signalled.

6.4. Simplification Procedures

In order to help minimize the amount of work that must be done by the theorem prover and to take advantage a certain specialized knowledge the formula generator has about the formulas it generates, the formula generator performs some simplification of the formulas it generates. Three types of simplification are performed as follows:

1. The formula generator can simplify certain very simple types of formulas to true. They are formulas whose consequent has the following form:

$$x \leq x$$
$$\text{bottom\_element} \leq x$$
$$x \leq \text{top\_element}$$

where $\leq$ is the partial ordering relation on security levels, x is any security level, bottom_element is the least security level in the partial ordering, and top_element is the greatest element in the partial ordering. Formulas of these forms are generated very frequently and simplifying them at the earliest possible stage significantly reduces the amount of work done by the formula generator and theorem prover, and also significantly reduces the amount of information that must be kept around. The formula generator prints out a count of the number of formulas it has simplified to true in this manner for each specification function it processes. The name of the INTERLISP function that carries out this simplification is TRIV.THM.

2. The second type of simplification involves taking each formula generated and performing a closure over the antecedents of certain important variables present in the consequent. This has the effect of identifying those antecedents that have relevance to the consequent. Only the antecedents so identified are included in the simplified formula. This simplification is done in the INTERLISP function MAINEXP.

3. The third type of simplification involves selecting antecedents for inclusion in the simplified formula by applying certain tests to each antecedent. A selected antecedent must pass at least one of the tests. The tests are:

   - The antecedent contains the consequent.

   - The antecedent contains a name present on a list of names provided by the user.

   - The antecedent contains an expression that is close in form to the consequent.

   - The antecedent contains another antecedent that has passed one of the above tests.

   This type of simplification is performed by the INTERLISP function TRIMANTS.

The formula generator produces two simplified versions of each formula it passes to the theorem prover. Note that the unsimplified version is not passed. The first simplified version involves simplifications of types 2 and 3 above. The second simplified version involves simplication of type 2 only. The theorem prover will first attempt to prove the first simplified version and, if it is unable to prove this version, will attempt to prove the second simplified version. If neither proof attempt succeeds, then the formula is considered unproven. Note that if the simplification of type 1 above is possible, the formula is known to be true and there is no need to pass the formula on to the theorem prover at all.

6.5. Augmentation of the Boyer-Moore Theorem Prover

The theorem prover part of the automated tools is based on the Boyer-Moore theorem prover [1]. However, certain additions were made to the basic theorem prover enable it to prove theorems of the type generated by the formula generator. These additions are enumerated below.

- The Boyer-Moore theorem prover was modified to accept lower case characters in its formulas. It also was augmented to accept the characters "!" (exclamation point) and "_" (underscore).

- All of the theorems listed in <u>A Computational Logic</u> [1] are known to the theorem prover as lemmas.

- Several SPECIAL operations are defined to the theorem prover, namely: =, ~=, ~, =>, IF, OR, and AND.

- Several axioms have been given to the theorem prover. These are:

  ```
         x INSET {x}
        x INSET {x, y}
        y INSET {x, y}
    x INSET UNION({x}, y)
    y INSET UNION(x, {y})
  ```

  where x and y are any identifiers.

- All the SPECIAL operators handled by the formula generator other than those which have been defined are declared as functions to the theorem prover.

What is surprising about this list is how small it is and how little the theorem prover needs to know about SPECIAL in order to prove the generated formulas. Of course, the sufficiency of this list depends upon the style in which the specifications are written and, therefore, more information about the operators of SPECIAL may have to be added in order to prove formulas generated for future uses. It is easy to write specifications that are secure and yet will not be proven by these tools as they are currently constituted; however, the more important issue is whether or not anyone is likely to write such specifications. In any case it is a simple task to increase the knowledge of the theorem prover by introducing new definitions or axioms. One must be careful about the introduction of new axioms, for they may have an adverse effect upon the mathematical soundness of the tools. One must also be careful to assure that new axioms or definitions about SPECIAL operations are consistent with the definition of SPECIAL.

7. Use of the Automated Tools

This section provides a brief introduction to the use of the automated tools. The tools do not provide a comprehensive solution to the problem of proving that a given specification describes a multilevel secure system, they merely provide assistance in performing some of the more laborious and lengthy computations. In order to produce a complete and sound proof of multilevel security, the user of these tools must understand precisely what the tools do and do not provide. The first part of this section establishes the context in which the use of the tools is meaningful. The second part of this section describes the use of the tools.

7.1. Setting up

The definition of multilevel security with respect to which proofs are done has been given above. Specifications are written in SPECIAL. A complete description of SPECIAL is given in the HDM Handbook, Volume II [4]. It will be assumed that the reader is familiar with these documents. A SPECIAL specification provides a description of the externally visible behavior of a system, i.e., a description of how the system responds to each possible external stimulus. The possible external stimuli are the invocations of the visible operation references (a visible operation together with a particular set of values for its arguments). The specification describes (1) how the internal state of the system changes when a particular visible operation reference is invoked, and (2) the value returned by the invocation of the operation reference. The multilevel security model requires that there be a set of values L which acts as security levels and that these values be partially ordered under some binary relation, named here as "$\leq$". In addition, there must be some mathematical function, K, that maps the union of the set of visible operation references and the set of hidden primitive V-function references (the state variables) into the set of security levels. All this information must be provided before a proof can be attempted.

Most of this information must be supplied explicitly to the tools, but some is supplied implicitly. For example, the specifications and the definitions of $\leq$ and K are supplied explicitly, whereas the set of security levels L is supplied implicitly and is the set of all values related by $\leq$ and in the range of K. The tools check the validity of some of the information supplied, but not all. For example, the syntactic legality of the specifications is checked, but the fact that the relation $\leq$ partially orders L is assumed. Therefore, presenting a relation to the tools to serve as $\leq$ that does not partially order L would invalidate any resulting proof. The validity of some of the information is checked by means of imposed convention. For example, the function K is assured to be a true mathematical function, because the value of K for any given function reference must be either an argument of the function reference or one of a predetermined set of constants.

The result of applying the multilevel security proof tools is a listing of the attempted proofs of a set of formulas. If the attempted proofs of all the formulas are successful, this implies that the specification is multilevel secure with respect to the given securitv levels. Note that the proof says nothing about how the security levels are interpreted or how access to them is controlled in a given implementation.

The current version of the tools is able to prove multilevel security only for specifications written in a particular subset of SPECIAL, and then only within the framework of the given model -- i.e., for suitable definitions of $\leq$, K, and L. The restrictions on SPECIAL are as follows:

- No recursive or mutually recursive definitions are permitted.
- The key words NEW, TYPECASE, and RESOURCE_ERROR may not be used.
- An expression may contain no more than one reference to a new value. A new value is either a quoted V-function reference, an EFFECTS_OF expression, or (in an OV-function) a return value reference. (Note that for the purposes of this restriction, the elements of a conjoined expression are considered to be separate expressions.)
- In the effects of an OV-function, the return value reference may occur only once.
- A new value reference may not occur in:
    * The qualification part of a LET, FORALL, EXISTS, SOME, or set expression.
    * The antecedent of an implication.
    * The boolean expression in an IF expression.
    * The range of a vector constructor.

If the specification consists of more than one module, the directed graph of external references between the module must have no loops.

7.2. Use

The multilevel security verification tools are part of the collection of tools used to support the Hierarchical Development Methodology (HDM). The multilevel security verifier accepts the information described above and generates a collection of formulas whose truth implies the multilevel security of the given specification with respect to the given L, $\leq$, and K. The theorem prover is then used to attempt to prove these formulas true The result is a file showing the formulas generated and the attempted proofs. The use of the tools described below is illustrated by the examples of Figures 7-2, 7-3, and 7-4. Figure 7-3 gives the terminal session for the application of the tools to the specification given in Figure 7-2. Figure 7-4 shows the output file produced by this application.

The multilevel security verifier is invoked by calling the HDM support tools. A more complete discussion of the tools is given in Volume II of the HDM Handbook [4]. On the SRI-KL system, these tools reside in the file NS.EXE in the directory HIER. They can be invoked from TOPS-20 command level by typing the command

```
    <HIER>NS
```

This is an Interlisp subsystem; once you see the Interlisp prompt character, you will be at Interlisp command level. The verifier assumes that the specification for the system to be proved secure consists of one or more SPECIAL modules and that the text for each module is in a file with name of the form

```
    <name>.SOURCE
```

where <name> is the name of the module. The verifier is invoked by typing "MLS" followed by the name of the module for which formulas are to be generated (see examples at the end of this document). If the specification consists of more than one module, then each module must be run through the verifier. If a module contains external references to another module, then the referenced module must be run through the verifier before the referencing module. Whenever a module is run through the verifier, a file with name <name>.TLINK is created at the user's option. This file contains information needed by the verifier when processing a module that externally references this module. The user must, therefore, request such a link file when processing a module that is so referenced.

```
MODULE security

TYPES
security_level: DESIGNATOR;

PARAMETERS
BOOLEAN lteq(security_level l1, l2);

ASSERTIONS
FORALL security_level l1: lteq(l1, l1);
FORALL security_level l1; security_level l2; security_level l3
       | lteq(l1, l2) AND lteq(l2, l3): lteq(l1, l3);
FORALL security_level l1; security_level l2
       | lteq(l1 ,l2) AND l1 ~= l2: ~lteq(l2,l1);

END_MODULE
```

Figure 7-1: Specification for module security

```
MODULE virtual_memory

    PARAMETERS

INTEGER max_seg_no, max_seg_index;


    EXTERNALREFS

    FROM security:
security_level: DESIGNATOR;
BOOLEAN lteq(security_level l1, l2);

    FUNCTIONS

VFUN contents(INTEGER segno, index; security_level sl)
         -> INTEGER c;
    HIDDEN;
    INITIALLY
       c = ?;

VFUN read(INTEGER segno, index; security_level sl)
         [security_level pl]
         -> INTEGER c;
    EXCEPTIONS
       segno < 0 OR segno > max_seg_no;
       ~lteq(sl, pl);
       contents(segno, index, sl) = ?;
    DERIVATION
       contents(segno, index, sl);

OFUN write(INTEGER segno, index, c; security_level sl)
          [security_level pl];
    EXCEPTIONS
       segno < 0 OR segno > max_seg_no;
       index < 0 OR index > max_seg_index;
       ~ lteq(pl, sl);
    EFFECTS
       'contents(segno, index, sl) = c;
       FORALL INTEGER i | i >= 0 AND i < index
                          AND contents(segno, i, sl) = ?:
          'contents(segno, i, sl) = 0;

END_MODULE
```

Figure 7-2: Specification for a virtual memory

```
@<hier>ns
SRI Specification Handler and Hierarchy Manager  (30-Dec-79)

Type DOC for documentation and NEWS for the latest news.
(<HIER>NS.EXE.8 . <LISP>LISP.EXE.132)
_MLS SECURITY
<FEIERTAG>SECURITY.SOURCE.2

Beginning generating formulas for module security
The partial order function for security levels is defined as:
   LessThanOrEqualTo(arg1 arg2) = lteq(arg1 arg2)
The bottom element is: low_level
The top element is: high_level
Link file? yes
<FEIERTAG>SECURITY.TLINK.1
END
```

Figure 7-3: Terminal output

```
_MLS VIRTUAL_MEMORY
<FEIERTAG>VIRTUAL_MEMORY.SOURCE.23

Beginning generating formulas for module virtual_memory
The partial order function for security levels is defined as:
   LessThanOrEqualTo(arg1 arg2) = ... lteq (arg1 arg2)
The bottom element is: ... low_level
The top element is: ... high_level
<FEIERTAG>SECURITY.TLINK.1
Beginning processing V-function contents
Which argument is the access level of contents (segno index sl) ? sl
Beginning processing V-function read
Which argument is the access level of read (segno index sl pl) ? pl
2 formulas generated.
Beginning processing of O-function write
Which argument is the access level of write (segno index c sl pl) ? pl
3 formulas generated.
1 trivial formulas resolved to true.
<FEIERTAG>VIRTUAL_MEMORY.FORMULAS.100021
Link file? ...Yes
<FEIERTAG>VIRTUAL_MEMORY.TLINK.1
Attempting to prove formulas for module VIRTUAL_MEMORY
lteq,LTEQ.REFLEXIVITY,LTEQ.TRANSITIVITY,
collecting lists
5126, 10246 free cells
low_level,LTEQ.BOTTOM,high_level,LTEQ.TOP,contents,max_seg_index,
max_seg_no,Beginning proving theorems for function read
Unable to prove theorem for EXCEPTION (= (contents segno index sl) ?)
1  formulas proved for function  read

Beginning proving theorems for function write
2  formulas proved for function  write

Beginning proving theorems for function read
Unable to prove theorem for EXCEPTION (= (contents segno index sl) ?)
0  formulas proved for function  read
Failed to prove theorems.
END
_LOGOUT)
@
```

Figure 7-3 - Terminal output (cont'd)

```
Declaring partial ordering for  lteq (arg1 arg2) .
Adding axioms:

_DCL(lteq (arg1 arg2) NIL)
lteq
(EQUAL (lteq A0017 A0017) T)

_ADD.AXIOM(LTEQ.REFLEXIVITY (REWRITE)
                            (EQUAL (lteq A0017 A0017) T)
                            NIL)
LTEQ.REFLEXIVITY
(IMPLIES (AND (lteq A0017 A0018) (lteq A0018 A0019)) (lteq A0017 A0019))

_ADD.AXIOM(LTEQ.TRANSITIVITY (REWRITE)
                             (IMPLIES (AND (lteq A0017 A0018)
                                           (lteq A0018 A0019))
                                      (lteq A0017 A0019))
                             NIL)
LTEQ.TRANSITIVITY
Declaring bottom element low_level with axiom:

_DCL(low_level NIL NIL)
low_level
(EQUAL (lteq (low_level) A0017) T)

_ADD.AXIOM(LTEQ.BOTTOM (REWRITE)
                       (EQUAL (lteq (low_level) A0017) T)
                       NIL)
LTEQ.BOTTOM
Declaring top element high_level with axiom:

_DCL(high_level NIL NIL)
high_level
(EQUAL (lteq A0017 (high_level)) T)

_ADD.AXIOM(LTEQ.TOP (REWRITE)
                    (EQUAL (lteq A0017 (high_level)) T)
                    NIL)
LTEQ.TOP
```

Figure 7-4: Proof file generated by tools

```
Declare contents (segno index sl)

_DCL(contents (segno index sl) NIL)
contents
Declare max_seg_index NIL

_DCL(max_seg_index NIL NIL)
max_seg_index
Declare max_seg_no NIL

_DCL(max_seg_no NIL NIL)
max_seg_no
```

Figure 7-4 - Proof file generated by tools (cont'd)

```
Module: virtual_memory
Function: read
EXCEPTION
(= (contents segno index sl)
   ?)

Proving:
(IMPLIES (lteq sl pl)
         (lteq sl pl))

This formula simplifies, clearly, to:

      (TRUE).

Q.E.D.

Load average during proof:  2.681496
Elapsed time:  2.506 seconds
CPU time (devoted to theorem proving):  .213 seconds
GC time:  0.0 seconds
IO time:  .136 seconds
CONSes consumed:  84
```

Figure 7-4 - Proof file generated by tools (cont'd)

```
Module: virtual_memory
Function: read
DERIVATION
(contents segno index sl)

Proving:
(IMPLIES (lteq sl pl)
         (lteq sl pl))

This conjecture simplifies, obviously, to:

      (TRUE).

Q.E.D.

Load average during proof:  2.681496
Elapsed time:  .234 seconds
CPU time (devoted to theorem proving):  .148 seconds
GC time:  0.0 seconds
IO time:  .069 seconds
CONSes consumed:  84
```

Figure 7-4 - Proof file generated by tools (cont´d)

```
Module: virtual_memory
Function: write
EFFECT
(= (NEWSTATE contents segno index sl)
   c)

Proving:
(IMPLIES (lteq pl sl)
         (lteq pl sl))

This simplifies, trivially, to:

      (TRUE).

Q.E.D.

Load average during proof:  2.6666
Elapsed time:  .259 seconds
CPU time (devoted to theorem proving):  .157 seconds
GC time:  0.0 seconds
IO time:  .067 seconds
CONSes consumed:  84
```

Figure 7-4 - Proof file generated by tools (cont´d)

```
Module: virtual_memory
Function: write
EFFECT
(FORALL ((¦ (INTEGER i)
            (AND (AND (>= i 0)
                      (< i index))
                 (= (contents segno i sl)
                    ?))))
        (= (NEWSTATE contents segno i sl)
           0))

Proving:
(IMPLIES (lteq pl sl)
         (lteq pl sl))

This formula simplifies, obviously, to:

      (TRUE).

Q.E.D.

Load average during proof:  2.6666
Elapsed time:  .273 seconds
CPU time (devoted to theorem proving):  .162 seconds
GC time:  0.0 seconds
IO time:  .068 seconds
CONSes consumed:  84

Events undone: (max_seg_no max_seg_index contents LTEQ.TOP high_level
LTEQ.BOTTOM low_level LTEQ.REFLEXIVITY LTEQ.TRANSITIVITY lteq)
Theorems proved.
```

Figure 7-4 - Proof file generated by tools (cont′d)

Once invoked, the verifier types the name of the source file. This signifies that the module has been successfully parsed. This is followed by a listing of type checking errors, if any, and the beginning of formula generation. The verifier then asks for the partial ordering relation. You must respond by typing an invocation of the partial ordering relation, i.e., the name of the partial ordering relation followed by its arguments in parentheses. The two arguments named arg1 and arg2 represent the two security levels being compared and the invocation you type must mean that the given relation is true is arg1 is less than or equal to arg2. In the example shown, the partial ordering relation is named "lteq", so the user types

```
lteq(arg1 arg2)
```

signifying that the value of lteq(arg1 arg2) is true if arg1 is less than or equal to arg2.

Next, the verifier asks for the name of the bottom element in the partial ordering. If the bottom element is needed to answer any subsequent questions of the verifier, its name must be provided at this point. Simply type the name of the bottom element followed by a carriage return. If you do not need to supply the name of the bottom element then simply type a carriage return. The same procedure holds for the top element of the partial ordering.

If the user does not respond to the requests for the names of the partial ordering relation, the bottom element, and the top element within some period of time, the verifier will use the information supplied by the user the last time the verifier was invoked in this session. In this case, the verifier will type out the information it is using preceded by "...".

As the verifier processes each function in the module, it will request that the user supply the value of the function K for the function it is processing. As stated above, the user must respond with the name of one of the parameters to the function followed by a carriage return. Alternatively, the user can respond with the name of the bottom element or the name of the top element if either was supplied in answer to the questions described above. In the case of a nonprimitive V-function, an O-function, or an OV-function, the user may respond with the name NONE. This signifies that this function is not part of the interface under consideration and that no formulas should be generated. For each function in the specification, the verifier will state the number of formulas generated from the specification of the function, if any, and the number of these generated formulas that it can immediately determine to be true. The formulas known to be true are removed from the list of generated formulas.

Once the verifier has completed generating formulas for a module, the user is asked if he wants a link file produced. As mentioned above, the user must respond "y" for "yes" if the verifier is to be run on a module that externally references the module being processed. A response of "n" for "no" means that no link file is produced.

The verifier then invokes the theorem prover to attempt to prove the formulas true. The verifier prints out information about any formula it is unable to prove true. If all formulas are proved then the verifier prints "Theorems proved". This indicates that the module being considered has a multilevel secure interface with respect to the information provided. The formulas and their attempted proofs are listed in a file of the form

```
<name>.PROOFS
```

where <name> is the name of the module. Operation of the verifier ends with typing of the word "END".

As mentioned above, the verifier remembers the names of the partial ordering relation, the bottom element, the top element, and all the security level arguments associated with the functions from one invocation to another within a single invocation of the HDM tools. However, it is possible to make the verifier remember this information permanently. This is done by typing the command MPF (make profile) while in the HDM tools. This command creates a file named MLS.PROFILE in your current directory that contains the current values of the partial ordering relation, the bottom element, and the top element. Whenever you subsequently invoke the HDM tools from this directory, this file will be read and the state of the verifier restored to this value. Remember that these defaults can be overridden by typing the desired response when the information is requested by the verifier.

7.3. Less Restrictive Model Formulation

It is possible to instruct the verification tool to prove some less restrictive security requirements. This is accomplished by supplying an optional argument to the MLS command as follows:

```
MLS <name> <option>
```

where <name> is, as before, the name of the specification module to be analyzed, and <option> is either not present or one of: SSO or *PO. If <option> is not present then the verifier works as described above and attempts to prove the full multilevel security model. If the <option> is SSO then the verifier attempts to prove Equation 2 plus the equation

$$v_1 \xrightarrow{f} v_2 \text{ AND } \sim(K(v_1) \leq K(f)) \Longrightarrow K(v_1) \leq K(v_2) \tag{5}$$

Specifications that obey Equations 2 and 5 are said to have the simple security property. If the <option> is *PO then the verifier attempts to prove Equation 4. Specifications that obey Equation 4 as said to have the *-property (pronounced star-property). These properties are meaningful because they enforce multilevel security above or below the level of the subject. For example, if a given specification has the simple security property, then a given subject can violate multilevel security below its own security level but cannot violate multilevel security above its security level.

7.4. Interpreting the Verifier's Output

In the example given above, the result of applying the verifier to the specifications is a set of proven theorems. However, most applications of the verifier to a specification are likely to be less successful. In these cases it will be necessary for the user to interpret the output from the verifier in order to ascertain why some formulas could not be proved and if the inability to prove a formula is due to a security violation in the specifications. Depending upon the complexity and size of the specifications this could be an easy or a rather painstaking task.

The output of the verifier consists of the messages it prints on the users terminal and the contents of the proofs file. The terminal output summarizes the success or lack of success of the verifier. The proofs file provides the information necessary to analyze the cause of a lack of success. The proof file lists all the commands given to the theorem prover and the response of the theorem prover to those commands. In general, the proof file gives a fairly thorough description of how it attempted to prove a particular formula and it is, therefore, fairly easy to detect why a particular formula could not be proved. However, the proof file is not as thorough in documenting the source of a formula and it can be difficult in some cases to pinpoint the difficulty in the specification that leads to the unprovable formula. This is not an inherent difficulty in the verifier, it is simply an area that has not been given much attention in this early stage of the verifier's development. This section provides the user with some advice on how to relate a formula found in the proof file to that part of the specification that gave rise to the formula.

The bulk of the proof file documents the attempted proof of the generated formulas. The beginning of the proof file documents some theorem prover commands necessary to set up the proof of the formulas. All functions used in the formulas must be known to the theorem prover. All the operators defined by SPECIAL have been previously defined or declared to the theorem prover. However, functions particular to a given specification must be declared. These functions include the primitive V-functions and the functicnal parameters of the specification. In addition the security level partial ordering function is declared. In order to inform the theorem prover about the partial ordering function the properties of reflexivity and transitivity on this partial ordering relation are introduced as axioms. (Note that the property of antisymmetry has not been included because it has not generally proved to be necessary.) Any global assertions of the specifications are also introduced as axioms to the theorem prover. Once these preliminaries are complete, the theorem prover attempts to prove the formulas. Finally, the theorem prover removes from its data base all the axioms and declarations derived from the given specification. The theorem prover is then ready to begin anew. The proof file provides complete documentation of this process.

For each formula which the theorem prover attempts to prove, the proof file records the name of the module from which the formula was generated, the name of the specification function from which the formula was generated, the type of paragraph from which the formula was generated (i.e., EXCEPTIONS, EFFECTS, DELAY, or DERIVATION), the exception, effect, delay clause, or derivation from which the formula was generated, the generated formula, and a description of the attempted proof. For a straightforward specification, determining which expression in the specification produces a particular formula is rather easy because the expression is given as part of the documentation. Relating the identifiers in the formula to identifiers in the specification is very easy also because they are identical or very similar (i.e., the identifier in the formula may have a disambiguating suffix). However, the part of the specification from which a given formula is generated and which is given in the proof file is the unexpanded text. The formula is derived from the text after all definitions, EXCEPTIONS_OF expressions, and EFFECTS_OF expressions have been fully expanded. In the cases of specifications that make extensive use of such expressions and where they are deeply nested, the formula generated may bear little obvious resemblance to the original unexpanded sou.ce expression. Also in specifications where the same name is used over and over again in different scopes, it may become difficult to find the identifier in the specification corresponding to a particular identifier in a formula. In these cases, the user may have to use heuristic pattern matching techniques or expand the specification by hand in order to determine precisely the source of a given formula. However, experience has shown that even in fairly complex specifications one quickly learns to recognize patterns in formulas and specifications and, after tracing back a few formulas, one can trace the others back fairly quickly.

Let us investigate the specification given in Figure 7-5. This is a slightly modified version of the specification given in Figure 7-2 above. The verifier is applied to this specification as shown in Figure 7-6 and part of the resulting proof file is shown in Figure 7-7. Figure 7-7 contains only that part of the proof file different from that given in Figure 7-4. Looking at this formula we see that it obviously cannot be proven because there is insufficient information.

We note that the untrue formula arises from the second exception to the function "read". This exception accesses the primitive V-function "contents(segno, index, sl)". This primitive V-function has the security level "sl". The security level of the "read" operation is "pl". There are no established conditions for relating these two levels. We must show that "sl $\leq$ pl" or in terms of the specification that "lteq(sl, pl). However, this is precisely the condition established by the third exception to "read". The obvious solution to the problem is to place the third exception before the second. Since the exceptions are evaluated in order of appearance, if the third exception is evaluated, the second exception must have been false.

```
MODULE virtual_memory

    PARAMETERS

INTEGER max_seg_no, max_seg_index;


    EXTERNALREFS

    FROM security:
security_level: DESIGNATOR;
BOOLEAN lteq(security_level l1, l2);

    FUNCTJONS

VFUN contents(INTEGER segno, index; security_level sl)
             -> INTEGER c;
    HIDDEN;
    INITIALLY
       c = ?;

VFUN read(INTEGER segno, index; security_level sl)
         [security_level pl]
         -> INTEGER c;
    EXCEPTIONS
       segno < 0 OR segno > max_seg_no;
       contents(segno, index, sl) = ?;
       ~lteq(sl, pl);
    DERIVATION
       contents(segno, index, sl);

OFUN write(INTEGER segno, index, c; security_level sl)
          [security_level pl];
    EXCEPTIONS
       segno < 0 OR segno > max_seg_no;
       index < 0 OR index > max_seg_index;
       ~ lteq(pl, sl);
    EFFECTS
       'contents(segno, index, sl) = c;
       FORALL INTEGER i ¦  i >= 0 AND i < index
                          AND contents(segno, i, sl) = ?:
          'contents(segno, i, sl) = 0;

END_MODULE
```

Figure 7-5: Specification for an insecure virtual memory

```
_MLS VIRTUAL_MEMORY
<FEIERTAG>VIRTUAL_MEMORY.SOURCE.24

Beginning generating formulas for module virtual_memory
The partial order function for security levels is defined as:
   LessThanOrEqualTo(arg1 arg2) = ... lteq (arg1 arg2)
The bottom element is: ... low_level
The top element is: ... high_level
<FEIERTAG>SECURITY.TLINK.1
Beginning processing V-function contents
Which argument is the access level of contents (segno index sl) ? ...sl
Beginning processing V-function read
Which argument is the access level of read (segno index sl pl) ? ...pl
2 formulas generated.
Beginning processing of O-function write
Which argument is the access level of write (segno index c sl pl) ? ...pl
3 formulas generated.
1 trivial formulas resolved to true.
<FEIERTAG>VIRTUAL_MEMORY.FORMULAS.100021
Link file? ...Yes
<FEIERTAG>VIRTUAL_MEMORY.TLINK.2
Attempting to prove formulas for module VIRTUAL_MEMORY
lteq,LTEQ.REFLEXIVITY,LTEQ.TRANSITIVITY,low_level,LTEQ.BOTTOM,high_level
,LTEQ.TOP,contents,max_seg_index,max_seg_no,
Beginning proving theorems for function read
2  formulas proved for function  read

Beginning proving theorems for function write
2  formulas proved for function  write
Theorems proved.
END
```

Figure 7-6: Terminal output for insecure specification

```
Module: virtual_memory
Function: read
EXCEPTION
(= (contents segno index sl)
   ?)

Proving:
(lteq sl pl)


     Name the conjecture *1.


     Since there is nothing to induct upon, the proof has





**********************************************************************

***                                                                ***

***                       F A I L E D !                            ***

***                                                                ***

**********************************************************************

Load average during proof:  3.810249
Elapsed time:  3.147 seconds
CPU time (devoted to theorem proving):  .332 seconds
GC time:  0.0 seconds
IO time:  .183 seconds
CONSes consumed:  146


Will later attempt to prove:
(lteq sl pl)
```

Figure 7-7: False formula generated from insecure specification

Therefore, if we interchange the second and third exceptions, when "contents(segno, index, sl) = ?" is encountered, then "~lteq(sl, pl)" must be false or "lteq(sl, pl)" must be true. Hence we can trivially show that the requirement generate from "contents(segno, index, sl) = ?", namely "lteq(sl, pl)" is true. We see this result in the formula (in Figure 7-4) generated from the correct specification of Figure 7-2.

8. Conclusion

This document has described a technique and automated tools that support the technique for proving that a the specification, written in SPECIAL, of a given system is multilevel secure. The automated tools are applicable to specifications written within certain guidelines given above, however, these guidelines are not overly restrictive. Even in the case of systems that are not completely multilevel secure, the technique and the tools are extremely useful in establishing that the only security violations are those that were intended. Use of these techniques and tools on several system specifications has exposed previously unknown security flaws in these systems thereby demonstrating the utility of the technique and tools.

One cannot, however, blindly apply the tools without proper understanding of their use. The tools can provide meaningful results only if they are provided with meaningful input. Providing improper security levels or an invalid partial order function renders the output of the tools meaningless. The tools are therefore just tools and provide meaningful assistance only if they are utilized properly as part of the overall proof technique.

It is also necessary to recall that the technique and the tools are useful only in proving the multilevel security of a specification. They do not guarantee that a specification is consistent or realizable. However, the technique is extremely valuable in accomplishing what has previously been a very difficult task.

9. Possible Future Enhancements

Although there is an unlimited number of possible improvements and enhancements that could be made to the technique and tools described above, there are a few areas where experience has shown that some modifications are important to significantly increasing the effectiveness of the technique. Following is a list of such areas for improvement in no particular order.

- With respect to the multilevel security verification technique, the semantics of SPECIAL is rigorously defined only within the code for the formula generator and in the definitions and axioms of the theorem prover. This is only a partial definition relating to those properties of SPECIAL that are relevant to verifying multilevel security. Other definitions exists in other tools. There is no assurance that any of these definitions are consistent, or that they embody the SPECIAL manual. Parts of the semantics of SPECIAL are not documented at all either formally or informally. This is not to say that all the different views of SPECIAL are grossly diverse, it simply means that there might be differences in some of the fine points. A complete and fairly rigorous definition of a specification language would be useful.

- The automated tools are significantly slower than they have to be. They take from 15 to 90 minutes of CPU time on a DEC KL-10 processor to process a system specification from moderate to high complexity. Although this is not unbearably slow, it means that the tools are hard to use on an interactive basis. Some fairly straightforward modifications could significantly improve the performance of the tools. For example, it would appear that a much less sophisticated theorem prover would suffice for proving the formulas that are generated. A simpler theorem prover could execute much more quickly. About two thirds of the time spent in the tools is spent in the theorem prover. Also, the formula generator is not currently block compiled. Certain parts of the operation of the formula generator could significantly benefit from block compiling. Attention to some of the less efficient parts of the tools could probably improve its performance by a factor of five.

- Very little attention has been paid to the interface of the user with the tools. The interface is rather clumsy. Also the user gets little help in analyzing the output from the tools. The tools have sufficient information to isolate security flaws very well and could very possibly produce a list of given specification with the offending subexpressions underlined. It is conceivable that the tools could provide the user with a list of possible corrections although this is somewhat more difficult.

- The automated verification tools currently handle only a subset of SPECIAL. It is possible for the tools to handle the entire SPECIAL language. However, the subset was chosen because it contained the most used portions of SPECIAL and because reasonable formulas could be generated and proved. As the remaining portions of SPECIAL are addressed by the tools, the dependency heuristics will need to become more complex and the formulas will be harder to prove. As the formulas become more complex and difficult to prove, the tools become slower and less effective for frequent use. Therefore, one must be careful that any effort to have the tools handle the remaining and infrequently used constructs does not render the tools so inefficient as to make them a burden. The existence of the current allowed subset of SPECIAL forces the user to utilize the tools most effectively.

REFERENCES

1. R.Boyer and J Strother Moore. A Computational Logic. Academic Press, 1979.

2. R.S.Boyer and J Strother Moore. A Theorem-Prover For Recursive Functions: A User's Manual. CSL-91, SRI International, 979.

3. R.J.Feiertag, K.N.Levitt,and L. Robinson. Proving Multilevel Security of a System Design. Proceedings of Sixth ACM Symposium on Operating Systems Principles, ACM, November, 1977.

4. B.Silverberg, L.Robinson, and K. Levitt. HDM Handbook, Volume II: The Languages and Tools of HDM. SRI International, June, 1979.

5. W. Teitelman. Interlisp Reference Manual. Xerox Palo Alto Research Center, 1978.